# A hybrid genetic algorithm for the distributed permutation flowshop scheduling problem

**Jian Gao, Rong Chen**[*]
*College of Information Science and Technology, Dalian Maritime University*
*Dalian, 116026, China.*

Received 12 December 2010
Accepted 31 January 2011

**Abstract**

Distributed Permutation Flowshop Scheduling Problem (DPFSP) is a newly proposed scheduling problem, which is a generalization of classical permutation flow shop scheduling problem. The DPFSP is NP-hard in general. It is in the early stages of studies on algorithms for solving this problem. In this paper, we propose a GA-based algorithm, denoted by GA_LS, for solving this problem with objective to minimize the maximum completion time. In the proposed GA_LS, crossover and mutation operators are designed to make it suitable for the representation of DPFSP solutions, where the set of partial job sequences is employed. Furthermore, GA_LS utilizes an efficient local search method to explore neighboring solutions. The local search method uses three proposed rules that move jobs within a factory or between two factories. Intensive experiments on the benchmark instances, extended from Taillard instances, are carried out. The results indicate that the proposed hybrid genetic algorithm can obtain better solutions than all the existing algorithms for the DPFSP, since it obtains better relative percentage deviation and differences of the results are also statistically significant. It is also seen that best-known solutions for most instances are updated by our algorithm. Moreover, we also show the efficiency of the GA_LS by comparing with similar genetic algorithms with the existing local search methods.

*Keywords*: Distributed scheduling; Permutation flowshop; Genetic algorithm; Local search

## 1. Introduction

The Permutation Flowshop Scheduling Problem (PFSP) is a widely investigated complex optimization problem. Especially, the permutation flowshop scheduling problem with makespan criterion has been investigated most frequently in the past decades.[1-5] Though the permutation flowshop scheduling problem with makespan criterion is relatively simple, it is still a hard combinatorial problem. It is reported that the problem is NP-Complete in the strong sense when the number of machines in the problem is larger than 3.[6] Given a PFSP, all jobs in a PFSP have to be processed in the same factory. Namely, the classical PFSP is based on the assumption that there is only one factory or shop. However, many modern companies have changed their manufacturing environments, where traditional single-factory environment is replaced by multi-factory environment and more factories are built to set up the environment[7]. It is reported[8] that the distributed environment can make companies achieve better product quality, lower production cost and lower management risks. Multi-factory companies will play a more important role in practice. Therefore, the DPFSP has been introduced by Naderi and Ruiz[9] recently. It is a generalization of the classical permutation flowshop scheduling problem, where a set of factories is combined with the classical problem and each job is allowed to be processed on one factory. The optimization criterion discussed in Ref. 9 is the minimization of the maximum completion time among all the factories. Note that there is an overlap between DPFSP and flow

[*]Corresponding author. *E-mail address*: rchen@dl.cn

shop with parallel machines (FSPM)[10] which handles a set of jobs with sets of parallel identical machines in some processing centers. But they differ from each other in terms of representation of solutions and applied algorithms because machines between centers are unrelated in FSPM.

Algorithms for classical PFSP have been deeply studied. Since Johnson published the first paper[11] about flowshop problem, there have been many publications on studying flowshop problems and various approaches to this problem have been proposed. In particular, there are a large mount of papers to deal with PFSP with makespan criterion. Exact searching algorithms, such as integer programming, branch & bound and backtracking, can obtain the optimized solutions. However, to some disappoint, those algorithms usually take a long CPU time to solve large PFSP because of the computational complexity of it. As stated in Ref 12, the best performing algorithms among those exact algorithms are able to solve instances with 20 jobs in maximum. Thus, those algorithms can only cope with small-scale scheduling problems. To solve large problem instances effectively for practical purposes, most researchers have to focus on developing heuristics that can find a near optimal solution in reasonable time. Some representative heuristics include: the index heuristic proposed by Palmer,[13] the CDS method proposed by Campbell et al.[14] and the NEH algorithm proposed by Nawaz et al.,[15] where the NEH algorithm is regarded as one of the most efficient heuristics among heuristic algorithms. Some newly proposed constructive methods were also published.[16,17] Improvements on NEH also emerged recently.[5,18] The heuristic algorithms can usually build a feasible solution in polynomial time, but the quality of solutions may be not satisfactory. Therefore, metaheuristic algorithms for the PFSP have been developed. An increasing number of papers on this topic are being published. Tabu search methods were proposed in earlier studies. For instance, Armentano and Ronconi proposed a tabu search algorithm for tardiness minimization[19] and a fast tabu search algorithm was presented by Grabowski and Wodecki for makespan criterion.[3] Simulated annealing and artificial immune algorithm were also used to solve the flowshop problems.[20,21] Genetic algorithms[12,22,23] and the ant colony optimization in the Rajendran and Ziegler's work[4] have been published in the last decade, as well as particle swarm optimization (PSO)[24-26] and differential evolution algorithm[27] in recent works. In addition, the hybrid metaheuristic algorithms have also been investigated recently.[24] Some algorithms mentioned above, such as GA and PSO, are combined with local search methods in order to enhance searching for local-optimal solution. Many experiments show those hybrid algorithms usually have good performance.

As stated in Ref. 9, studies involving the DPFSP are rare and other kinds of distributed scheduling problems in the literature are in their infancy. Since DPFSP is a newly proposed scheduling problem, algorithms for DPFSP are only discussed in Ref 9. They are the extensions of algorithms for PFSP. The work mainly includes algorithms of mixed integer linear programming and heuristics. Six mixed integer linear programming models have been investigated and implemented on highly optimized CPLEX11.1 package. Their performance was analyzed carefully. From the experimental results, it can be seen that only small instances (16 jobs and 4 factories) were tested by those exact algorithms. While to solve large instances, 12 heuristics approaches, derived from well-known existing ones for PFSP, were presented, where two alternative rules for job assignments are combined with those heuristics: one locates the job to the factory with the lowest partial makespan; the other one tries all possible positions of all the factories for a job and places the job in the position that has the lowest makespan. Moreover, the work also presented a local search approach for the DPFSP, called Variable Neighborhood Descent (VND). It is a simple version of variable neighborhood search. The approach starts from the solution of NEH heuristic method, moves jobs in each factory, and then moves jobs from the factory with maximal makespan to other factories. There are two criteria that are employed to accept job movements. Thus, two VND algorithms are referred as VND(a) and VND(b) by using the two acceptance criteria respectively. Experiments on large instances indicate that VND(a) has the best performance among all the algorithms in Ref. 9, though the total CPU times are longer than others.

Genetic Algorithm (GA), first developed by Holland,[28] is a well-known adaptive heuristic search method. It is also a bio-inspired algorithm and is used to solve many optimization problems.[12,29,30] It starts searching with the initial population where individuals are distributed in the search space. It works by iterating the three operators that are selection, crossover and mutation. Better solutions in the current generation will be reserved by the selection operation. Crossover, the next operator, is used to produce offspring for the next generation. Furthermore, mutation is used to escape from local minima by randomly modifying some offspring. GA has shown a good performance due to its global search ability compared with heuristic approaches and local search algorithms. GA also performs well on large instances of PFSP.[12] However, to the best of our knowledge, there is little work published on GA methods or other evolutional methods for the distributed PFSP. In this work, we propose a hybrid meta-heuristic algorithm based on GA to solve distributed PFSP instances. Also an efficient local search is incorporated into the GA algorithm for improving the performance. The entire proposed algorithm is denoted by GA_LS. The features of our GA_LS can be summarized as follows: since the representation of solutions for DPFSP is a set of job sequences, to make operators in GA suitable for the representation, crossover and mutation methods are proposed; the local search is carried out on some selected individuals with the aim of finding better solutions, where methods for job movements within a factory, job movements and exchanges between factories are designed. We carefully analyze the results of experiments on the benchmark instances, from which we can see that the proposed GA_LS obtains much better solutions than the existing heuristic algorithms and updates most best-known solutions. We also perform experiments to compare the efficiency of GA_LS and genetic algorithms using the local search methods VND(a) and VND(b). The results indicate that GA_LS is the most efficient algorithm among them.

## 2. The distributed permutation flowshop scheduling problem

The permutation flowshop scheduling problem can be described as follows:[3] each of $n$ jobs from the set $J=\{1,2,\ldots,n\}$ has to be processed on $m$ machines in the order of $1,2,\ldots,m$. Job $j$, $j\in J$, consists of a sequence of $m$ operations $O_{j1},O_{j2},\ldots,O_{jm}$; operation $O_{jk}$ corresponds to the processing of job $j$ on machine $k$ and is associated with a processing time $p_{jk}$. Each machine can only process one job at a time. Each job can be processed only on one machine at a time. All jobs are uninterrupted. The objective is to find a sequence of the jobs so that the given criterion is optimized.[9] In this paper, we consider the maximum completion time or makespan as the criterion.

Let $\pi$ be a sequence of all jobs and $C(j,k)$ denotes the completion time of $O_{jk}$. So we can calculate $C(j,k)$ by the following formulas.[31]

$$\pi=\{j_1,j_2,\ldots,j_n\}$$
$$C(j_1,1)=p_{j_1 1}$$
$$C(j_i,1)=C(j_{i-1},1)+p_{j_i 1} \text{ for } i=2,\ldots,n$$
$$C(j_i,k)=C(j_i,k-1)+p_{j_i k} \text{ for } k=2,\ldots,m$$
$$C(j_i,k)=\max\{C(j_{i-1},k),C(j_i,k-1)\}+p_{j_i k} \text{ for } i=2,\cdots,n\ ;\ k=2,\ldots,m$$
$$C_{max}(\pi)=C(n,m)$$

where $C_{max}(\pi)$ is the makespan. The task of solving a permutation flowshop scheduling problem is to find a $\pi$ so that $C_{max}(\pi)$ is minimized.

The DPFSP can be defined as follows[9]: $n$ jobs from the set $J=\{1,2,\ldots,n\}$ have to be processed on $F$ factories, where each factory $f\in G=\{1,\ldots,F\}$ contains the same set of $m$ machines, which is same as the PFSP. All factories are able to process all jobs. When a job $j$ is assigned to a factory $f$, it can not be transferred to another factory and all operations of it can only be processed at factory $f$. Each operation $O_{jk}$ is associated with a processing time $p_{jk}$. Assume that this processing time of the operation is available for all factories. Namely, the processing time of $O_{jk}$ in one factory is same as these of other factories. A schedule of jobs is a set of job sequences, denoted by $\prod$. $\prod$ contains $F$ job sequences. The intersection of any two job sequences is empty and the union of all job sequences is the set $J$. The makespan of a schedule $\prod$ is defined as the maximum makespan among all factories, and can be formulated as follows.

$$C_{max}(\textstyle\prod)=\max\{C_{max}(\pi_f)\} \text{ for } f\in G$$

where $\pi_f$ denotes the job sequence of the $f$-th factory. The goal of a DPFSP is to find the minimal makespan of the DPFSP.

## 3. The proposed genetic algorithms

### 3.1 *Solution representation and initialization*

When coding a solution of PFSP, permutation of jobs is commonly used to represent a feasible scheduling. A permutation is a processing order of jobs in the machines. We can calculate the makespan by calculating complete time of each job according to the permutation. For a DPFSP, the representation can be naturally extended to a set of job sequences, one for a factory. This method is also employed in Ref. 9 for their VND algorithms. It is complete as the set of sequences can represent all the possible solutions.

Initialization for genetic algorithms can be achieved by randomly generating the sequences of jobs. But in order to accelerate optimization of GA and ensure a faster convergence to good solutions, the initialization with NEH2 and VND(a)[9] is carried out. The solutions obtained by NEH2 and VND(a) are assigned to randomly selected individuals while other individuals are initialized by generating job sequences randomly.

## 3.2 *Design of operators*

Selection mechanism used in our GA is the method in the classical genetic algorithms, where individuals are ranked according to the fitness and then selected. The mutation operator is also designed for our algorithm. It exchanges some pairs of jobs randomly, and the number of pairs is chosen randomly but smaller than the half of the total number of jobs. The fitness of individuals is defined as follows:

$$fitness = (1 / C_{max})^3$$

Because differences of $C_{max}$ for individuals may be very small, we use $(1/C_{max})^3$ to enlarge the differences.

Crossover operator is to generate two new individuals that are probably of good fitness from two selected individuals. There are many crossover operators that have been presented in the literature. For example, order crossover was proposed by Davis.[32] In Ref. 23, Murata et al. discussed one-point crossover and two-point crossover. As a mixed operator, one-point order crossover, which combines ideas of the one point crossover and the order crossover,[33] was introduced, as well as two-point order crossover. Similar job order crossover and similar block order crossover were proposed by Ruiz et al. in Ref. 12. Because of the difference between the representation of DPFSP and that of classical PFSP, the crossover operator has to be rewritten. Our crossover operator is designed using a simple method. It is quite similar to the one-point (OP) crossover, which is quite easy to implement and extend to the representation of DPFSP.

Our crossover operator selects points randomly for all factories of the second parent, which are used to divide job sequences of the parent. The set of jobs on the right sides is denoted by *R*, and jobs in *R* will be removed from the first parent. The remaining jobs in each sequence of the first parent are placed in the order of their appearance and inherited to the child. The right sides of the second parent are conjoined to the corresponding factories of the child. On the other hand, the other child is produced with the above method by exchanging the role of two parents.

An example of the crossover operation is illustrated in Fig. l. The set *R* of Parent 1 is composed of Jobs 1, 2, 3 and 5. Jobs 4, 6 and 7 are inherited from Parent 1 to the Child 1, and Jobs 5, 3 and 1 from the first factory of Parent 2 are conjoined to the first factory of Child 1 while Job 2 is conjoined to the second factory.

Note that job sequences in a child obtained by the crossover operator may be not balanced. That is, some sequences may contain many more jobs than others, which probably causes the total makespan increases greatly. But it does not mean the crossover operator is not effective because local search is employed in our hybrid GA to adjust job sequences, and jobs in long sequences will be moved to other factories by the some methods discussed in the next subsection.

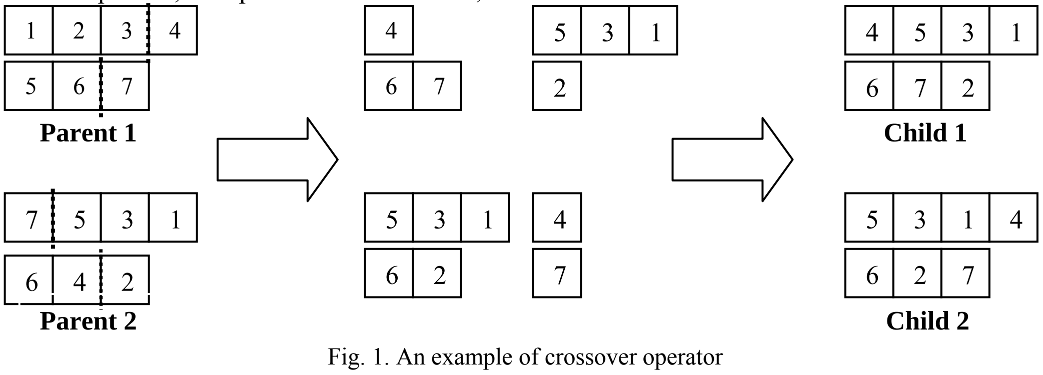

Fig. 1. An example of crossover operator

### 3.3 *Local search*

To overcome the disadvantage mentioned above and enhance searching in the local area, we design local search method for our genetic algorithm. The procedures of local search are widely used in evolution algorithms. It is highly capable at accelerating convergence and finding better solutions. In Ref. 9, VND(a) and VND(b) have been investigated for DPFSP, which are on the basis of variable neighborhood search. Job insertion is employed for a single factory, and jobs in the factory with the maximal makespan will be tried to relocated at all possible positions of other factories, from which the best movement will be performed.

```
Insertion_Jobs
let $f$ be the factory to be improved
foreach job $j_1$ in the factory $f$
        select $j_2$ randomly ($j_1 \neq j_2$)
        remove $j_1$, $j_2$ from $f$
        find the best position of $j_1$ in $f$ and assign it to the position
        find the best position of $j_2$ in $f$ and assign it to the position
end
Move_Jobs
flag $\leftarrow$ true
while flag do
        flag $\leftarrow$ false
        let $f_{max}$ be the factory with maximum makespan and $C_{max}$ be the makespan
        let $f_{min}$ be the factory with minimum makespan.
        foreach job $j$ in $f_{max}$
                find the best position of $j$ from all possible positions in $f_{min}$
                if new makespan of $f_{min}$ is smaller than $C_{max}$
                        remove $j$ from $f_{max}$ and assign it to the best position in $f_{min}$
                        flag $\leftarrow$ true
                        break
        end
end
Exchange_Jobs
flag $\leftarrow$ true
while flag do
        flag $\leftarrow$ false
        let $f_{max}$ be the factory with maximum makespan and $C_{max}$ be the makespan
        let $f_{min}$ be the factory with minimum makespan.
        foreach job $j$ in $f_{max}$
                try to exchange $j$ with each job in $f_{min}$
                find the best exchange and denote the job in $f_{min}$ by $j_1$
                if both new makespans of $f_{min}$ and $f_{max}$ are smaller than $C_{max}$
                        exchange $j$ and $j_1$.
                        flag $\leftarrow$ true
                        break
        end
end
```

Fig. 2. The procedure of Insertion_Jobs, Move_Jobs and Exchange_Jobs.

Motivated by VND, we propose three new job movement methods: Insertion_Jobs, Exchange_Jobs and Move_Jobs. Insertion_Jobs is designed for a single factory while the others are used to move jobs between factories. The procedures of them are shown in Fig. 2. In the procedure of Insertion_Jobs, each job $j_1$ of the factory is selected accompanying with a random selected job $j_2$, and the two jobs are removed from the factory. After that, $j_1$ is reassigned at the best positions and then $j_2$. In the procedure of Move_Jobs, the factories with the maximal makespan and the minimal makespan are found, denoted by $f_{max}$ and $f_{min}$. We only consider moving jobs in $f_{max}$ to $f_{min}$ instead of all other factories because it will update $f_{max}$ and $f_{min}$ if a better movement is performed. For each job $j_1$ in $f_{max}$, all possible positions in $f_{min}$ will be tried and moved to the best position if the movement improves the makespan of the DPFSP. Different from the method in Ref. 9, it will break to perfrom the next iteration when an improvement is made. Otherwise, it continues trying the next job. The iteration will stop if all the jobs are tried but no improvement can be made. In the procedure of Exchange_Jobs, similar to Move_Jobs, $f_{max}$ and $f_{min}$ are found first. A job $j_1$ in $f_{max}$ is exchanged with each job in $f_{min}$, and best exchange will be performed if it can improve the makespan. Then new $f_{max}$ and $f_{min}$ will be found for starting the next iteration. Iteration continues until it can not be improved.

```
Local search
flag ← true
foreach factory f
    perform Insertion_Jobs on f
end
while flag do
    perform Move_Jobs
    perform Exchange_Jobs
    if there is any factory f changed
        perform Insertion_Jobs on f
    else
        flag ← false
    end
end
```

Fig. 3. The procedure of local search.

The procedure of local search is indicated in Fig. 3, it is similar with VND method in Ref. 9, except the three new job movement methods. Each factory is processed by Insertion_Jobs first, and then Exchange_Jobs and Move_Jobs performed by marking the changed factories for further improvement by Insertion_Jobs. Consider Child 1 in Example 1, our local search may perform as the following steps. 2 factories are processed by Insertion_Jobs first. Suppose the result is that $f_1$={5,4,3,1} and $f_2$ does not change, and $f_1$ generates the overall $C_{max}$. Thus, Move_Jobs tries to move each job in $f_1$ to the 4 positions in $f_2$, and Exchange_Jobs tries to exchange each job in $f_1$ with 3 jobs in $f_2$. Let us also suppose that Job 3 is inserted into $f_2$ after Job 7, then $f_1$={5,4,1} and $f_2$={6,7,3,2} has to be processed by Insertion_Jobs again because they are changed. The procedure of local search will be stop in a local minimum when Move_Jobs and Exchange_Jobs cannot make any change. It is noted that we can use the accelerations developed by Taillard[34] when implementing the three sub-procedure. It can decrease CPU time greatly especially for large-scale instances.

### 3.4 *An overview of GA_LS*

Now we give the entire algorithm of GA_LS. Fig. 4 depicts the main procedure.

```
GA_LS
while stopping criterion is not satisfied do
    evaluate fitness
    apply local search on the best individual and a randomly selected
    individual
    update the best solution
    apply selection operator
    assign the best solution to a randomly selected individual
    apply crossover operator
    apply mutation operator
end
```

Fig. 4. The procedure of GA_LS

Local search is performed within the each iteration after fitness computation with the aim at balancing jobs sequences in the individuals and finding a minimum solution in the local area. However, we only take the best individual and a random chosen individual to perform the proposed local search, as some individuals produced by crossover and mutation operators may be not good enough. Note that it may converge too fast if local search is only performed on the best solutions. Moreover, the best solution is copied to the offspring by assigning it to a random individual.

## 4. Experiments

To evaluate the performance of the proposed GA_LS, intensive computational experiments are carried out. In this section, the DPFSP benchmark problems are employed to test algorithms. The benchmark is available at http://soa.iti.es, where only large-scale instances are considered in the paper, as the number of jobs is up to 16 and the number of machines is only 5 at most in small-scale instances. The set of large-scale instances is extended from the benchmark of Taillard by adding the number of factories $F$ from {2,3,4,5,6,7}. The Taillard instances are composed of 12 combinations of $n \times m$, and for each combination there are 10 different instances. Each instance in DPFSP benchmark is combined with six values to yield six instances of DPFSP benchmark, so the number of total instances reaches 720.

In order to show the effectiveness and efficiency, comparison between existing heuristics discussed in Ref.

9 and the proposed GA_LS is carried out. Furthermore, we also compare the proposed GA_LS with the genetic algorithms associated with existing local search methods. Those genetic algorithms are tested with same parameters, where population size is set to 20, the maximum number of iterations is 100. They are decided on the basis of the results of preliminary computational observations. When the population size is up to 40, the convergence speed of those algorithms may decrease to a low value, while early convergence occurs when the size is set to a small value (e.g. 10). For most instances, when the number of iterations exceeds 100, the makespan is improved slightly or converges to the local-optimal solution, so the maximum number of iterations is set to 100. The mutation probability is 0.1. The above algorithms are incorporated in a C++ program and implemented within VC++6.0. We perform all experiments on a PC with an Intel Core Duo 2.4GHz CPU, 2GB RAM, running Windows XP.

## 4.1 *Comparison of best-known solutions*

The best solutions of the benchmark instances at http://soa.iti.es are obtained by the heuristic approaches mentioned in Ref. 9. As our GA_LS algorithm is initialized by NEH and VND algorithms, GA_LS updates most of the best solutions greatly. In this section, we perform the GA_LS algorithm for all the 720 instances 10 times, and record the best solutions. Then, we evaluate our results of best solutions produced by GA_LS, and compare them with best-published results at http://soa.iti.es. Among all the 720 instances, GA_LS updates 692 best solutions. Relative percentage deviation is employed to measure the results.

$$RPD = \frac{alg - opt}{opt} \times 100$$

where *opt* is the best solution published and *alg* stands for the best solution obtained by GA_LS.

Table 1 summarizes the results grouped by *F* as well as the results grouped by each combination of *n* and *m*. From Table 1, it can be seen that GA_LS provides better solutions than all the algorithms mentioned in Ref. 9, as *RPD* values for all groups are minus. We can also see the total average *RPD* is -2.22%. GA_LS improves solutions with *F*=2,3,4,5 a bit better than these with *F*=6,7, where the average *RPD* values for *F*=2,3,4,5 are better than the total average value. This is because GA_LS got solutions with same objective as (or worse than) the best-published results when solving some small instances with *F*=6,7 and *n*=20. Among instances with *n*=20, there are 12 instances unimproved out of 30 instances when *F* is set to 7 and 7 instances when *F* is set to 6, while the number is 1,2,3,3 for *F*=2,3,4,5 respectively. However, it is worth noting that all the best solutions of instances with *n*≥50 are improved by GA_LS. In addition, among those 28 unimproved instances, only 6 solutions are worse than the best-published results. Moreover, Table 1 also demonstrates GA_LS can get minus *RPD* for each combination of *n* and *m*.

Table 1. Average relative percentage deviation (*RPD*) of GA_LS

| *F* | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| *RPD* | -2.26 | -2.48 | -2.36 | -2.32 | -2.08 | -1.85 |
| *n*×*m* | 20×5 | 20×10 | 20×20 | 50×5 | 50×10 | 50×20 |
| *RPD* | -2.36 | -1.91 | -1.38 | -2.75 | -2.92 | -2.44 |
| *n*×*m* | 100×5 | 100×10 | 100×20 | 200×10 | 200×20 | 500×20 |
| *RPD* | -1.87 | -2.72 | -2.41 | -1.99 | -2.14 | -1.77 |
| Average | | | | | | **-2.22** |

## 4.2 *Compare to other algorithms*

In this subsection, we replace the best-published solutions by our results produced by all the experiments in this paper including experiments in the next subsection. Many heuristic approaches have been tested in the Ref. 9 by Naderi and Ruiz. As those heuristics are extensions of the well-known existing heuristics for solving regular PFSP, the effectiveness of them mainly depends on that of the corresponding original algorithms. We only consider these four existing algorithms NEH1, NEH2, VND(a) and VND(b) in our experiments. The reason of this is those algorithms have far better performance than other heuristic algorithms evaluated in Ref. 9. Because there is no other meta-heuristics published to solve DPFSP, we implement two similar genetic algorithms to show our genetic algorithm with the presented local search method is efficient, where the operations of GA are same as the proposed one but they combine VND(a) and VND(b) as local search methods respectively. They are denoted by GA_VND(a) and GA_VND(b). All the parameters in these two algorithms are same as GA_LS.

To compare the performance of these algorithms mentioned in the former paragraph, NEH1, NEH2, VND(a) and VND(b) are carried out once for all 720 instances, while GA_VND(a) and GA_VND(b) are carried out 10 times for all 720 instances. Average CPU time, average makespan, best makespan and worst makespan of GAs are recorded.

First, we analyze *RPD* of the above experiment. Table 2 indicates the results grouped by *F*. Average *RPD* values of 7 algorithms are listed. At the bottom of the table, the total average values of all *F* are reported as well. As can be seen, GA_LS outperforms all the other algorithms. Heuristic and local search algorithms have bigger *RPD* than genetic algorithms. It is not surprise that GA algorithms perform better than the NEH2 and VND(a) because of the initialization. However, among the three GAs, GA_LS performs best, while improvements of GA_VND(b) are quite limited as the method of VND(b) is not considerable compared with VND(a). Moreover, it is interesting to note that the *RPD* of the worst solutions got by GA_LS is even better than the corresponding *RPD* of the best solutions produced by GA_VND(a), where we can find 0.91% and 1.41% for the total average *RPD* respectively. The best solution found by GA_LS produces an average *RPD* of 0.23%. Though GA_LS updates most of the best-known solutions, the *RPD* is not very close to 0%. Note that the new best solutions employed here are the best solutions obtained by both GAs with 100 iterations and GAs with limited CPU times (discussed in the next subsection).

Next, we give the *RPD* results grouped by the combination of *n* and *m*, as well. Table 3 shows the results. From the table, similar conclusion can be drawn as from Table 2. For each combination, GA_LS performs best regardless of the average, best and worst *RPDs*.

We also need to check whether the differences in Table 2 and Table 3 made by those algorithms are statistically significant, which can help us to draw a better picture of the results. Three hypotheses (normality, homocedasticity and independence of the residuals) are checked and satisfied. Fig. 5 shows the ANOVA of the results, where mean plot with Tukey HSD intervals at 99% confidence level for the algorithm factor is depicted. From Fig. 5, we can clearly observe that the GA_LS has a very good performance overcoming all the remaining methods.

Next, we analyze CPU time of the experiment. The results grouped by *F* are listed in Table 4, where CPU time is in millisecond. Heuristic methods have the total average CPU time below 50ms while GAs take more time, where average times are larger than 2 seconds and 10.17 seconds for the algorithm GA_LS. The largest CPU time 158.8 seconds is found for solving the instance Ta_120_2 by GA_LS, where 500 jobs and 20 machines have to be scheduled. Though they needed far more time than heuristic methods needed, the average CPU times of them can be accepted in practice use. We can also see CPU times decrease as the *F* grows. This is consistent with the results produced by heuristic methods. Hence, the results indicate that the instance becomes easier to solve when *F* grows.

Among the three GAs, GA_LS needs the longest CPU time, where it is about 2 times larger than CPU time needed by GA_VND(a). So the local search method presented takes quite long time to find better solutions. To give a fair comparison between GAs, we conduct the experiments in the next subsection.

Table 2. Average relative percentage deviation (*RPD*) of algorithms grouped by *F*

| *F* | Algorithms | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | NEH1 | NEH2 | VND(a) | VND(b) | GA_VND(a) | | | GA_VND(b) | | | GA_LS | | |
| | | | | | Average | Best | Worst | Average | Best | Worst | Average | Best | Worst |
| 2 | 5.47 | 3.72 | 2.64 | 3.09 | 1.58 | 1.18 | 1.98 | 2.17 | 1.79 | 2.49 | 0.58 | 0.17 | 1.01 |
| 3 | 6.38 | 3.83 | 2.94 | 3.33 | 1.92 | 1.59 | 2.26 | 2.58 | 2.30 | 2.80 | 0.52 | 0.16 | 0.88 |
| 4 | 7.08 | 3.93 | 2.88 | 3.48 | 1.86 | 1.57 | 2.18 | 2.65 | 2.44 | 2.81 | 0.61 | 0.27 | 0.97 |
| 5 | 7.03 | 3.52 | 2.81 | 3.27 | 1.84 | 1.55 | 2.11 | 2.67 | 2.52 | 2.79 | 0.57 | 0.23 | 0.86 |
| 6 | 7.16 | 3.45 | 2.71 | 3.24 | 1.72 | 1.44 | 2.00 | 2.58 | 2.46 | 2.65 | 0.60 | 0.29 | 0.90 |
| 7 | 6.98 | 2.93 | 2.21 | 2.76 | 1.38 | 1.14 | 1.60 | 2.17 | 2.10 | 2.20 | 0.55 | 0.27 | 0.84 |
| Average | 6.68 | 3.56 | 2.70 | 3.19 | 1.71 | 1.41 | 2.02 | 2.47 | 2.27 | 2.62 | 0.57 | 0.23 | 0.91 |

Table 3. Average relative percentage deviation (RPD) of algorithms grouped by *n* and *m*

| $n\times m$ | Algorithms | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | NEH1 | NEH2 | VND(a) | VND(b) | GA_VND(a) | | | GA_VND(b) | | | GA_LS | | |
| | | | | | Average | Best | Worst | Average | Best | Worst | Average | Best | Worst |
| 20×5 | 7.18 | 4.58 | 3.77 | 4.40 | 2.49 | 2.46 | 2.54 | 3.66 | 3.66 | 3.66 | 0.82 | 0.74 | 0.87 |
| 20×10 | 6.11 | 3.39 | 2.79 | 3.05 | 1.66 | 1.54 | 1.75 | 2.71 | 2.67 | 2.73 | 0.75 | 0.57 | 0.94 |
| 20×20 | 4.53 | 2.72 | 2.00 | 2.60 | 1.37 | 1.34 | 1.42 | 1.86 | 1.84 | 1.93 | 0.60 | 0.45 | 0.74 |
| 50×5 | 9.54 | 4.23 | 3.36 | 3.93 | 2.43 | 2.28 | 2.55 | 3.20 | 3.12 | 3.23 | 0.48 | 0.24 | 0.74 |
| 50×10 | 8.13 | 4.35 | 3.42 | 4.02 | 2.26 | 1.86 | 2.59 | 3.11 | 2.90 | 3.26 | 0.74 | 0.13 | 1.33 |
| 50×20 | 6.38 | 3.81 | 2.78 | 3.37 | 1.90 | 1.55 | 2.23 | 2.50 | 2.32 | 2.65 | 0.64 | 0.11 | 1.14 |
| 100×5 | 7.67 | 2.80 | 2.07 | 2.49 | 1.38 | 1.09 | 1.67 | 1.98 | 1.91 | 2.04 | 0.44 | 0.10 | 0.80 |
| 100×10 | 7.80 | 3.78 | 2.81 | 3.33 | 1.84 | 1.39 | 2.28 | 2.51 | 2.20 | 2.76 | 0.60 | 0.09 | 1.11 |
| 100×20 | 6.05 | 3.64 | 2.70 | 3.19 | 1.58 | 1.07 | 2.13 | 2.34 | 1.91 | 2.67 | 0.56 | 0.12 | 1.03 |
| 200×10 | 6.75 | 3.13 | 2.36 | 2.77 | 1.32 | 0.83 | 1.88 | 2.12 | 1.74 | 2.34 | 0.43 | 0.10 | 0.80 |
| 200×20 | 5.51 | 3.51 | 2.40 | 2.89 | 1.33 | 0.85 | 1.83 | 2.03 | 1.60 | 2.36 | 0.44 | 0.08 | 0.80 |
| 500×20 | 4.56 | 2.82 | 1.90 | 2.29 | 1.01 | 0.69 | 1.37 | 1.60 | 1.33 | 1.84 | 0.34 | 0.07 | 0.61 |

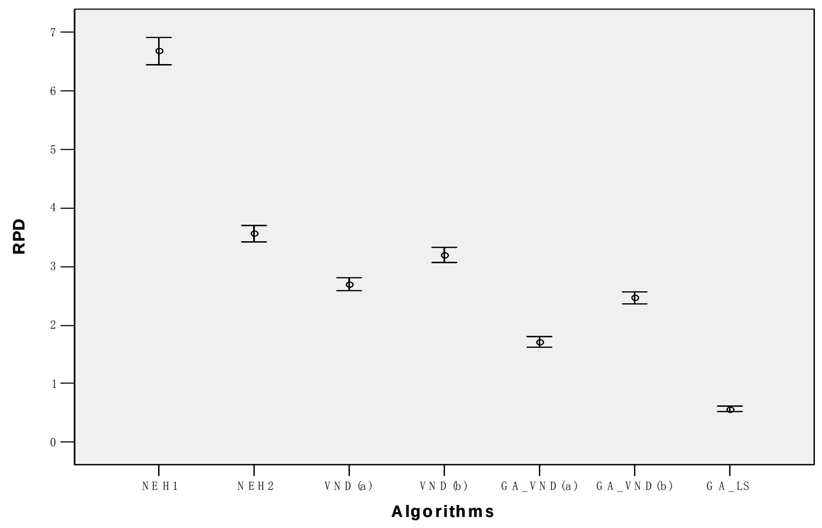

Fig. 5. Tukey HSD intervals at 99% confidence level for the algorithm factor

Table 4. Average CPU times of algorithms grouped by *F* (ms)

| *F* | Algorithms | | | | | | |
|---|---|---|---|---|---|---|---|
| | NEH1 | NEH2 | VND(a) | VND(b) | GA_VND(a) | GA_VND(b) | GA_LS |
| 2 | 3.61 | 7.12 | 73.72 | 49.35 | 4,899.94 | 3,776.00 | 15,539.58 |
| 3 | 2.37 | 7.44 | 50.17 | 36.53 | 3,668.66 | 3,043.78 | 13,179.13 |
| 4 | 1.92 | 7.66 | 40.87 | 27.57 | 3,168.10 | 2,586.34 | 10,529.33 |
| 5 | 1.58 | 7.80 | 32.74 | 22.35 | 2,756.56 | 2,107.91 | 8,558.90 |
| 6 | 1.30 | 7.91 | 28.15 | 19.46 | 2,474.34 | 1,820.40 | 7,148.80 |
| 7 | 1.15 | 8.06 | 22.62 | 17.60 | 2,018.05 | 1,479.86 | 6,089.73 |
| Average | **1.99** | **7.66** | **41.38** | **28.81** | **3,164.27** | **2,469.05** | **10,174.24** |

## 4.3 *Comparison of GAs with same CPU times*

Taking into account differences in CPU times, we record average CPU times of the three genetic algorithms for all the 720 instances. To conduct fair experiments, we execute the three algorithms replacing the stopping criterion of iteration times by time limits. As each instance has three CPU times of the three genetic algorithms, there are three groups of CPU times that can be used as time limits. We employ each group as the time limits and carry out those algorithms so that those algorithms are executed with the same CPU times. Thus, three groups of experiments are performed, where each instance is solved using each algorithm by 10 runs. Average *RPD* values for best, worst and average makespan of solutions are analyzed grouped by *F*, and the results are listed in Table 5, 6 and 7, as well as the total average *RPD* values.

From the three tables, we can see that GA_LS performs best regardless of CPU times cost. With the help of the proposed local search method, our genetic algorithm has a fast convergence to good solutions. Since GA_LS produces best *RPD* for each *F*, it can solve instances with different *f* efficiently. We also observe that even the *RPD* values of worst solutions obtained by GA_LS are smaller than *RPD* values of best solutions yielded by others regardless of the time limits used.

Table 5. Average RPD of genetic algorithms using CPU times of GA_LS as time limits

| *F* | Algorithms | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | GA_VND(a) | | | GA_VND(b) | | | GA_LS | | |
| | Average | Best | Worst | Average | Best | Worst | Average | Best | Worst |
| 2 | 1.55 | 1.05 | 2.12 | 2.14 | 1.67 | 2.52 | 0.49 | 0.14 | 0.94 |
| 3 | 1.80 | 1.28 | 2.27 | 2.55 | 2.14 | 2.87 | 0.53 | 0.15 | 0.91 |
| 4 | 1.78 | 1.32 | 2.25 | 2.62 | 2.27 | 2.86 | 0.60 | 0.26 | 0.95 |
| 5 | 1.67 | 1.28 | 2.08 | 2.61 | 2.35 | 2.78 | 0.55 | 0.21 | 0.90 |
| 6 | 1.66 | 1.27 | 2.05 | 2.57 | 2.39 | 2.68 | 0.65 | 0.28 | 1.04 |
| 7 | 1.29 | 0.95 | 1.62 | 2.14 | 1.99 | 2.20 | 0.52 | 0.25 | 0.85 |
| Average | **1.62** | **1.19** | **2.06** | **2.44** | **2.14** | **2.65** | **0.56** | **0.22** | **0.93** |

Table 6. Average RPD of genetic algorithms using CPU times of GA_VND(a) as time limits

| *F* | Algorithms | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | GA_VND(a) | | | GA_VND(b) | | | GA_LS | | |
| | Average | Best | Worst | Average | Best | Worst | Average | Best | Worst |
| 2 | 1.59 | 1.28 | 1.92 | 2.17 | 1.87 | 2.45 | 0.86 | 0.60 | 1.15 |
| 3 | 1.83 | 1.50 | 2.21 | 2.62 | 2.36 | 2.82 | 0.82 | 0.55 | 1.09 |
| 4 | 1.85 | 1.56 | 2.17 | 2.68 | 2.43 | 2.84 | 0.92 | 0.59 | 1.20 |
| 5 | 1.79 | 1.50 | 2.09 | 2.69 | 2.52 | 2.79 | 0.91 | 0.64 | 1.16 |
| 6 | 1.72 | 1.47 | 1.95 | 2.56 | 2.41 | 2.66 | 0.90 | 0.63 | 1.14 |
| 7 | 1.45 | 1.21 | 1.65 | 2.15 | 2.06 | 2.20 | 0.73 | 0.50 | 0.94 |
| Average | **1.70** | **1.42** | **2.00** | **2.48** | **2.28** | **2.63** | **0.86** | **0.59** | **1.11** |

Table 7. Average RPD of genetic algorithms using CPU times of GA_VND(b) as time limits

| *F* | Algorithms | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | GA_VND(a) | | | GA_VND(b) | | | GA_LS | | |
| | Average | Best | Worst | Average | Best | Worst | Average | Best | Worst |
| 2 | 1.65 | 1.29 | 1.98 | 2.15 | 1.84 | 2.44 | 0.86 | 0.61 | 1.15 |
| 3 | 1.93 | 1.65 | 2.26 | 2.58 | 2.32 | 2.79 | 0.97 | 0.74 | 1.24 |
| 4 | 1.86 | 1.60 | 2.17 | 2.66 | 2.45 | 2.81 | 0.93 | 0.66 | 1.22 |

Table 7(continued). Average RPD of genetic algorithms using CPU times of GA_VND(b) as time limits

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 1.79 | 1.55 | 2.05 | 2.68 | 2.54 | 2.79 | 0.88 | 0.65 | 1.13 |
| 6 | 1.74 | 1.45 | 2.00 | 2.58 | 2.46 | 2.66 | 0.88 | 0.65 | 1.11 |
| 7 | 1.48 | 1.29 | 1.73 | 2.15 | 2.07 | 2.19 | 0.74 | 0.52 | 0.96 |
| Average | **1.74** | **1.47** | **2.03** | **2.47** | **2.28** | **2.61** | **0.88** | **0.64** | **1.13** |

## 5. Conclusions

The distributed permutation flowshop scheduling problem is a newly proposed scheduling problem, which is in the set of NP-hard. Several heuristic and local search algorithms have been presented in Ref. 9, as well as MIP methods that can only solve small-scale instances though they are exact methods. There is little attention on meta-heuristic algorithms for DPFSP. Meanwhile, meta-heuristic algorithms play an important role for classical PFSP, and a great amount of works have been published on this topic. In this paper, we proposed a hybrid genetic algorithm with local search for minimizing the makespan of DPFSP. Our work can be summarized as follows: a genetic algorithm has been deeply studied, where the crossover operator was designed by extending the one-point crossover for general GAs; local search method was developed, where three local search operations were proposed by improving the existing VND method. The intensive experimental results revealed that our GA_LS performed much better than other algorithms in the literature. It can find solutions with better quality and even updated the best-known solutions of most benchmark instances. The experimental results also revealed that the genetic algorithm with the proposed local search method performs better than genetic algorithms combined with the VND methods. Experiments with same stopping criterions were carried out, where same iteration times and same CPU times were used as stopping criterions respectively. The results demonstrated GA_LS has the best *RPD* values compared with those genetic algorithms. A weakness of the proposed genetic algorithm is that when solving some small-scale instances that *n* is 20, it did not perform very well, since there are some instances whose best solutions can not be updated. But it has satisfactory performance on other instances ($n \geq 50$). Furthermore, since only simple crossover operator was investigated in our work, a future work is to propose and study other more efficient crossover operators for DPFSP with the aim at improving performance of genetic algorithms. We can see that meta-heuristics have achieved great successes on classical PFSPs, so designing other meta-heuristics to solve DPFSPs is also a necessary future work.

## Acknowledgements

We would like to thank the anonymous reviews for their valuable comments which have helped us to improve this paper. This work is partially supported by the National Natural Science Foundation of China under Grant No. 60775028, the Major Projects of Technology Bureau of Dalian No.2007A14GXD42, and IT Industry Development of Jilin Province.

## References

1. E. Vallada, R. Ruiz. Cooperative metaheuristics for the permutation flowshop scheduling problem. *European Journal of Operational Research*. 193(2) (2009) 365–376. doi:10.1016/j.ejor.2007.11.049
2. S. F. Rad, R. Ruiz, N. Boroojerdiana. New high performing heuristics for minimizing makespan in permutation flowshops. *Omega—International Journal of Management Science*. 37 (2009) 331–345. doi:10.1016/j.omega.2007.02.002
3. J. Grabowski, M. Wodecki. A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. *Computers & Operations Research*. 31(11) (2004) 1891–1909.
4. C. Rajendran, H. Ziegler. Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/ total flowtime of jobs. *European Journal of Operational Research*. 155 (2004) 426–438.
5. P.J. Kalczynski, J. Kamburowski. On the NEH heuristic for minimizing the makespan in permutation flow shops. *Omega—International Journal of Management Science*. 35 (2007) 53 – 60. doi:10.1016/j.omega.2005.03.003
6. M.R.Garey, D.S. Johnson, R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*. 1(2) (1976)117–129.
7. Felix T. S. Chan, S. H. Chung, P. L. Y. Chan: An adaptive genetic algorithm with dominated genes for distributed scheduling problems. *Expert Syst. Appl.* 29(2) (2005) 364–371. doi:10.1016/j.eswa.2005.04.009
8. H. Z. Jia, A. Y. C. Nee, J. Y. H. Fuh, Y.F. Zhang. A modified genetic algorithm for distributed scheduling

problems. *Journal of Intelligent Manufacturing*. 14(3–4) (2003) 351–362.

9. B. Naderi, R. Ruiz: The distributed permutation flowshop scheduling problem. *Computers & Operations Research* 37(4) (2010) 754–768. doi:10.1016/j.cor.2009.06.019
10. E. Nowicki, C. Smutnicki. The flow shop with parallel machines: A tabu search approach. *European Journal of Operational Research*. 106(2-3) (1998) 226-253. doi:10.1016/S0377-2217(97)00260-9
11. S. M. Johnson, Optimal two-and three-stage production schedules with setup times included. *Naval Research. Logistics Quarterly.* 1(1) (1954) 61–68.
12. E. Vallada, R. Ruiz. Genetic algorithms with path relinking for the minimum tardiness permutation flowshop problem. *Omega—International Journal of Management Science.* 38 (2010) 57–67. doi:10.1016/j.omega.2009.04.002
13. D.S. Palmer. Sequencing jobs through a multi-stage process in the minimum total time: a quick method of obtaining a near optimum. *Operational Research Quarterly*. 16(1) (1965) 101–107.
14. H.G. Campbell, R.A. Dudek, M.L. Smith. Heuristic algorithm for N-job, M-machine sequencing problem. *Management Science Series B—Application*. 16(10) (1970) 630–637.
15. M. Nawaz, Jr. E.E. Enscore, and I. Ham. A Heuristic Algorithm for the m-Machine, n-Job Flow-shop Sequencing Problem .*Omega—International Journal of Management Science*. 11(1) (1983) 91–95.
16. C. Koulamas. A new constructive heuristic for the flowshop scheduling problem. *European Journal of Operational Research*. 105 (1998) 66–71.
17. X. P. Li, Y .X. Wang, C. Wu. Heuristic algorithms for large flowshop scheduling problems. *In: Proceedings of the 5th world congress on intelligent control and automation.* (Hangzhou, China; 2004) pp. 2999–3003.
18. X. Dong, H. Huang, P. Chen. An improved NEH-based heuristic for the permutation flowshop problem. *Computers & Operations Research* 35(12) (2008) 3962–3968. doi:10.1016/j.cor.2007.05.005
19. V. Armentano, D. Ronconi. Tabu search for total tardiness minimization in flow-shop scheduling problems. *Computers & Operations Research*. 26 (1999) 219–235.
20. S. Parthasarathy, C. Rajendran. A simulated annealing heuristic for scheduling to minimize mean weighted tardiness in a flowshop with sequence-dependent setup times of jobs—a case study. *Production Planning & Control*. 8(5) (1997) 475–483.
21. O. Engin, C. Kahraman, M. K. Yilmaz. A New Artificial Immune System Algorithm for Multiobjective Fuzzy Flow Shop Problems. *International Journal of Computational Intelligence Systems*. 2(3) (2009) 236–247. doi:10.2991/ijcis.2009.2.3.5
22. C. Kahraman, O. Engin, I. Kaya, M. K. Yilmaz. An application of effective genetic algorithms for Solving Hybrid Flow Shop Scheduling Problems. *International Journal of Computational Intelligence Systems*. 1(2) (2008) 134–147. doi:10.2991/ijcis.2008.1.2.4
23. T. Murata, H. Ishibuchi and H. Tanaka. Genetic algorithms for flowshop scheduling problems. *Computers & Industrial Engineering*. 30(4) (1996) 1061–1071.
24. B. Liu, L. Wang, Y. Jin. An Effective PSO-Based Memetic Algorithm for Flow Shop Scheduling. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*. 37(1) (2007) 18–27.
25. B. Liu, L. Wang, Y. Jin. An effective hybrid PSO-based algorithm for flow shop scheduling with limited buffers. *Computers & Operations Research*. 35(9) (2008) 2791–2806. doi:10.1016/j.cor.2006.12.013
26. B. Li, L. Wang, B. Liu. An Effective PSO-Based Hybrid Algorithm for Multiobjective Permutation Flow Shop Scheduling. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*. 38(4) (2008) 818–831.
27. G. Onwubolu, D. Davendra. Scheduling flowshops using differential evolution algorithm. *European Journal of Operational Research*. 171 (2006) 674–692. doi:10.1016/j.ejor.2004.08.043
28. J.H. Holland. *Adaptation in natural and artificial system.* (Ann Arbor, Michigan, The University of Michigan Press, 1975)
29. R.S. Kumar, N. Alagumurthi. Integrated total cost and Tolerance Optimization with Genetic Algorithm. *International Journal of Computational Intelligence Systems*. 3(3) (2010) 325–333. doi:10.2991/ijcis.2010.3.3.8
30. P.G. Kumar. Fuzzy Classifier Design using Modified Genetic Algorithm. *International Journal of Computational Intelligence Systems*. 3(3) (2010) 334–342. doi:10.2991/ijcis.2010.3.3.9
31. C. R. Reeves. A genetic algorithm for flowshop sequencing. *Computers & Operations Research*. 22(1) (1995) 5-13.
32. L. Davis, Applying adaptative algorithms to epistatic domains, *in: Proceedings of the International joint conference on artificial intelligence*. (1985) pp. 162 – 164.
33. Z. Michalewicz. *Genetic algorithms + data structures = evolution programs*, 3rd ed., (Berlin, Heidelberg: Springer; 1996)
34. E. Taillard. Some efficient heuristic methods for the flow-shop sequencing problem. *European Journal of Operational Research.* 47(1) (1990) 65–74.